# CS 161: Homework 2

Submit via Gradescope by 3pm (PST), January 27, 2017.

**Instructions:** Please answer the following questions to the best of your ability. Unless otherwise indicated, provide full and rigorous proofs and include all relevant calculations. When writing proofs, please strive for clarity and brevity (in that order). Please see the course website for submission instructions (including Gradescope submission pass-code) and collaboration policy. Cite any sources that you use. You may discuss these problems with at most two other students, though you must write your own solutions, in your own words.

This homework contains a refresher on matrix operations, and more practice with the design and analysis of divide-and-conquer algorithms. (If you haven't done much with matrices, or its been a while since you've multiplied matrices, don't worry, this will not turn into a linear algebra class.)

1. **Matrix Operations** Matrices and linear algebra occur everywhere in computer science, especially in machine learning and optimization. In this problem we start by refreshing our memory of matrix operations. For each of the following parts, just give the answer, no need to show your work. (I suggest you do the calculations by hand, not by typing them into matlab....)

   (a) (1 point) Compute the following matrix sum: $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix} + \begin{bmatrix}1 & 2 & 1\\1 & 3 & 1\\1 & 4 & 1\end{bmatrix}$

   (b) (1 point) For the matrix $A = \begin{bmatrix}1 & 2\\4 & 5\\7 & 8\end{bmatrix}$, what is the *transpose* of $A$, denoted $A^T$?

   (c) (1 point) Compute the following matrix-vector product: $\begin{bmatrix}1 & 0 & 1\\0 & 1 & 1\\-1 & 1 & 2\end{bmatrix} \cdot \begin{bmatrix}3\\-2\\2\end{bmatrix}$

   (d) (1 point) Compute the following matrix product: $\begin{bmatrix}1 & 0 & 1\\0 & 1 & 1\\-1 & 1 & 2\end{bmatrix} \cdot \begin{bmatrix}3 & 1\\1 & 1\\1 & 1\end{bmatrix}$

2. The ability to efficiently compute matrix operations is central to the success of many machine learning pipelines. For example, the fact that your phone can do speech recognition without requiring an enormous battery-pack is largely because of extremely efficient algorithms for computing matrix products, and the products of matrices and vectors. In fact, the core of many signal-processing computations (including speech recognition), relies on the existence of special $n \times n$ matrices that have the property that you can multiply the matrix and a (length $n$) vector in time $O(n \log n)$. This is remarkable because even writing down the matrix explicitly would require $n^2$ time!!! In this problem, we will see one example of such a matrix. [Your solution to this problem will closely resemble the main idea behind the "Fast Fourier Transform", which some of you might have already encountered, and some of you will likely encounter in the near future.] Consider the following recursive definition for a class of matrices:

   $A_0 = [1]$, and for $i \geq 1$, we have

   $$A_i = \begin{bmatrix}A_{i-1} & 2A_{i-1}\\-A_{i-1} & 3A_{i-1}\end{bmatrix}.$$

Hence $A_1 = \begin{bmatrix} 1 & 2 \\ -1 & 3 \end{bmatrix}$, $A_2 = \begin{bmatrix} 1 & 2 & 2 & 4 \\ -1 & 3 & -2 & 6 \\ -1 & -2 & 3 & 6 \\ 1 & -3 & -3 & 9 \end{bmatrix}$, etc.
(Before proceeding, make sure you understand why this definition yields the claimed matrices $A_2$ and $A_3$.) First, observe that $A_i$ is a $2^i \times 2^i$ sized matrix, and hence even writing down $A_i$ requires $2^{2i}$ numbers.

(a) (1 point) Consider the problem of multiplying $A_i$ by a vector of length $N = 2^i$. Using the naive algorithm for matrix-vector multiplication, explain why this calculation would require $\Theta(N^2)$ basic operations, where a "basic operation" is either a sum or product of two numbers.

(b) (4 points) Using the recursive structure of $A_i$, define an algorithm for computing the product of $A_i$ and any length $N$ vector, $v$, such that your algorithm has the following property: for any integer $M = 2^j$, if $T(M)$ denotes the number of basic operations required to multiply $A_j$ and a length $M$ vector, then $T(M) = 2T(M/2) + O(M)$. Your answer for this part of the problem should include clear pseudo-code for your algorithm, and a proof of correctness that your algorithm computes the claimed matrix-vector product. [Hint: Consider partitioning the vector $v$ into two vectors $s$ and $t$ where $s$ consists of the first $N/2$ entries of $v$, and $t$ consists of the remaining entries of $v$. How can you represent the product $A_i v$ in terms of the quantities $A_{i-1}s$ and $A_{i-1}t$?]

(c) (1 points) Prove that the number of basic operations required by your algorithm satisfies the recurrence $T(M) = 2T(M/2) + O(M)$ where $T(M)$ denotes the number of basic operations required to multiply $A_j$ and a length $M = 2^j$ vector.

(d) (2 points) Analyze the above recurrence $T(M) = 2T(M/2) + O(M)$, and $T(1) = 0$ to show that $T(M) = O(M \log M)$. [Note that $T(1) = 0$ since computing the product of $A_1 = 1$ and any length 1 vector requires zero operations—we just return the vector. Feel free to use the "Master Method".]

3. **Sorting a Somewhat-Sorted Array**

(a) (2 points) What is the *best case* run-time of MergeSort? That is, find the largest function $g(n)$ such that for *every* array of length $n$, MergeSort takes at least $\Omega(g(n))$ operations to sort the array.

(b) (3 points) Define an array to be "$k$-somewhat-sorted" if it is possible to remove $k$ elements from the array and obtain a sorted array. Suppose you are given a $k$-somewhat-sorted array of length $n$, but you don't know $k$. Design an algorithm that runs in time $O(n)$ and takes as input a length $n$ "$k$-somewhat sorted array", and identifies a set of $O(k)$ elements (i.e. indices into the input array) with the property that after removing those elements, the array is sorted. Your solution should contain both the pseudo-code, a clear and succinct proof of the correctness of your algorithm (this can just be 2 sentences!), and a clear proof that the runtime is $O(n)$.

(c) (2 points) Give an algorithm that sorts a $k$-somewhat-sorted array of length $n$ in time $O(n + k\log(k))$ time. Your solution should contain both the pseudo-code, a clear and succinct proof of the correctness of your algorithm (this can just be 2 sentences!), and a clear proof that the runtime is $O(n + k \log k)$ [Hint: leverage the result from the previous part of the problem. Even if you didn't solve part (b), you can still do this part, assuming the existence of a solution to the previous part.]

4. **Having the Time of Your Life.** Suppose you decide to look through your journals and determine when the best time of your life was. Conveniently, your journals contain a numerical score for every day since kindergarten, quantifying the awesomeness of that day; e.g. 3/12/1989: "-1" (lost lunch money and was hungry), 3/13/1989: "+2.5' (Stacy from the school bus sat next to me),. . . ,5/22/2016: "+2.3" (made the bold decision to teach CS161 in Winter'17...nearly the same tingling combination of excitement and terror as that time Stacy sat next to me all those years ago), . . . , 1/19/2017: "-0.7"

(slightly sick and up late writing pset for 161), etc. You wish to determine when the best time of your life is/was, as defined by the contiguous time period with the largest total "awesomeness" score. For clarity, suppose your daily awesomeness scores are recorded in a length $n$ array $A = A[1], A[2], \ldots, A[n]$, where $A[i]$ is the awesomeness of the $i$th day since you started keeping records.

(a) (2 points) Provide a basic brute-force approach algorithm for calculating the best time-period of your life and analyze its runtime in terms of $n$. (Clearly describe the algorithm with either pseudo-code or one or two sentences, then prove your runtime claim with at most one sentence.)

(b) (2 points) A friend recommends that you use a divide-and-conquer approach for this problem so you divide $A$ into two halves and recursively compute the largest possible sum of any contiguous subarray for each half. How you would merge these two results in order to get a result for $A$? Clearly describe this divide-and-conquer recursive algorithm, and analyze its runtime. [Hint: in the merging step, do you need to consider subarrays that span the two halves?]

(c) (2 points) You return to your friend (Stacy) seeking advice on improving your result from part (b). She cryptically mentions the concepts of array prefixes and array suffixes: a prefix of $A$ is a subarray $P$ which starts at $A[1]$. Similarly a suffix of $A$ is a subarray $S$ which ends at $A[n]$. Using this hint, describe a divide-and-conquer algorithm for this problem that improves on the runtime of the algorithm from part (b). For full credit your improved algorithm should take total time $O(n)$, and follow the divide-and-conquer approach. (Later in the course we may see a *different* approach to this problem that would also yield an $O(n)$ time algorithm.) [Hint: consider having your recursive algorithm return more information than just a single subarray, namely consider returning information that helps the "merge" part of the previous algorithm be more efficient.]